Scalability vs. Performance

Daniel M. Pressel
U.S. Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5066

Keywords
supercomputer, high performance computing, parallel programming

Abstract

In the ideal world, the performance of a program running on a supercomputer would always be proportional to the peak speed of the system being used. Furthermore, the program would always achieve a high percentage of peak (e.g., 50% or better). In the real world, this is frequently not the case. Therefore, it is important to distinguish between the following five concepts:

(1) performance (run time),
(2) ideal speedup,
(3) hard scalability (fixed problem size speedup),
(4) soft scalability (scaled speedup), and
(5) throughput (how long it takes to run a collection of jobs).

This paper addresses these concepts and explains their meanings and differences. Hopefully, this will allow readers to evaluate the behavior of programs and computer systems, and most importantly, to evaluate their own expectations for running a program on a particular system or class of systems.

Examples, which demonstrate these concepts, are drawn from a variety of projects and include both problems from multiple computational technology areas (CTAs) and results from outside of the Department of Defense (DOD). In some cases, there will also be theoretical arguments to help better explain the issues.[*],[†]

1. Introduction

In the ideal world, the performance of a program running on a supercomputer would always be proportional to the peak speed of the system being used. Furthermore, the program would always achieve a high percentage of peak (e.g., 50% or better). In the real world, this is frequently

[*] This work was made possible by the grant of computer time by the DOD High Performance Computing Modernization (HPCM) Program. Additionally, it is largely based on work that was funded as part of the Common High Performance Computing Software Support Initiative (CHSSI) administered by the DOD HPCM Program.
[†] All items in bold type are defined in the glossary.

not the case. Therefore, it is important to study and discuss performance metrics for parallel systems and programming. Two important uses of these metrics are:

(1) the evaluation of the behavior of programs and computer systems and

(2) the evaluation of expectations for running a program on a particular system or class of systems.

The metrics that will be discussed in this paper are:

(1) performance (run time),
(2) ideal speedup,
(3) hard scalability (fixed problem size speedup),
(4) soft scalability (scaled speedup), and
(5) throughput (how long it takes to run a collection of jobs).

The discussion of these metrics will include a mixture of theoretical analysis and experimental results. The experimental results will come from a variety of disciplines but, in all cases, will involve real codes (e.g., no benchmarks) with representative data sets.

2. Performance

Most users are primarily interested in the following issues:

(1) the ability of the computer system to run their job;
(2) the correctness of the results;
(3) how fast does the job run once it starts running;
(4) how long will it take a series of jobs to complete; and
(5) when will the system start running their jobs.

The first, second, and fifth of these issues are beyond the scope of this paper. The fourth topic will be discussed in section 6. Performance can be quantified as:

$$\text{Performance} = \frac{\text{Theoretical Peak}}{\text{Performance}} \times \frac{\text{Algorithmic}}{\text{Efficiency}} \times \frac{\text{Serial}}{\text{Efficiency}} \times \frac{\text{Parallel}}{\text{Efficiency}} .$$

For many jobs, one can specify either a minimum acceptable level of performance and/or a desirable range for the performance. This need not preclude the achievement of even higher levels of performance. However, there may be resource allocation issues that favor sticking to the desirable range for the performance. What is important to note is that the program with the highest level of parallel efficiency may not be the program with the highest level of algorithmic efficiency[*]

[*] Algorithmic efficiency is a concept that can be difficult to measure in an absolute sense. However, it can generally be quantified in a relative sense (e.g., the number of floating point operations required to obtain a solution to a particular problem at a specified level of precision), and, in most cases, that is sufficient.

and vice versa. Furthermore, the history of parallel computing contains numerous examples of systems that would scale well, but on which it was notoriously difficult to obtain high levels of serial performance (e.g., the Thinking Machines CM2/CM200, many systems containing the Intel i860 microprocessor, and the Cray T3D [Bailey 1993; Simon and Dagum 1991; Simon et al. 1994; Bailey and Simon 1992; Oberlin 1999]). Therefore, it can be seen that all of the terms in this equation actively contribute to the delivered level of performance. This is a very different point of view from those who stress issues such as the following:

(1) The peak level of performance.

(2) The performance of a machine when running the unlimited size LINPACK benchmark (a benchmark that tends to have a high correlation with the peak speed of a system).

(3) That so long as a system is highly scalable with an efficient interconnect, one can "always" overcome a performance problem by using more processors (Simon et al. 1994).

Instead, what may be needed are combinations of programs and systems to run them on that provide an acceptable range of performance (preferably measured in run time, as opposed to **MFLOPS**) for a reasonable range of problem sizes and/or complexities. For example, if two programs can achieve similar results with similar levels of performance, for an acceptable range of problem sizes, then it is unimportant if the combination of program A and machine A has limited scalability past 64 processors and no scalability past 128 processors, while the combination of program B and machine B has good scalability to hundreds of processors. One might ask, how can this be? Some of the rationale behind this statement are:

(1) If the combination of program B and machine B needs the scalability just to match the performance of program A and machine A then, at best, program B and machine B are equal to program A and machine A.

(2) If one needs high levels of scalability to match another system's performance, then the cost effectiveness of the system must be considered.

(3) Scalability well beyond the planned size of a system is of primarily theoretical value.

(4) On most systems, most users have limited allocations and/or limited job priorities. Therefore, the user may find it difficult to use more than a certain number of processors at one time. Again, this results in unlimited levels of scalability to be primarily of theoretical value.

Of course, it is also possible that the combination of program B and machine B is not only more scalable, but also performs at least as well as program A and machine A on a per processor basis. In such a case, there may be a strong reason for favoring the combination of program B and machine B.

The following excerpts should help to demonstrate this point:

"One of the most intriguing aspects of linear elliptic boundary value problems (BVPs) is their relationship to probability.
.
.
.
It is obvious that this algorithm faithfully implements the collection of statistics implied in equation 2 in an "embarrassingly" parallel fashion. ... It also makes little difference if we implement this algorithm on a shared or distributed memory machine (or a loosely coupled group of workstations) since there is no interprocessor communication until the statistics are centrally collected.
.
.
.
It is well known that these Monte Carlo methods are much inferior to many deterministic methods for these types of problems" (Mascagni 1990).

Additional material on this topic can be found in Singh et al. (1998) and Wang and Tafti (1997).

3. Ideal Speedup

Frequently, it is necessary to predict the performance of a program for a fixed problem size when larger numbers of processors are used. In other cases, one needs to consider the relative merits of running two programs at once (each using half of the processors) vs. running the same programs sequentially (each using all of the processors). Questions such as these lead to the concept of *ideal speedup*.

The most commonly used definition for ideal speedup is that the speed at which the program runs on a particular machine is proportional to the number of processors being used. Returning to the concepts discussed in section 2, this is equivalent to saying that the parallel efficiency should be 100%.

Clearly from the standpoint of efficiency, unless the parallel efficiency is 100%, it is more efficient to run two programs at once, rather than running them sequentially. However, there are frequently other concerns (e.g., memory requirements and/or minimum performance requirements) that may outweigh this consideration.

There are many reasons why a program will not show linear speedup. Many of these have to do with limitations in the parallelization effort and/or inefficiencies in the hardware. As such, they are considered to be the cause for deviations from ideality. These topics will be readdressed later in this paper. However, one can argue that for some *algorithms*, and in particular, for some approaches to parallelizing those algorithms, that linear speedup is not the ideal speedup. Probably the most common example of this occurs when parallelizing a loop with M iterations when using N processors. If M is within about an order of magnitude of N, then the ideal speedup takes on the appearance of a staircase. This can best be seen in Table 1.

Table 1. Predicted Speedup for a Loop With 15 Units of Parallelism

Number of Processors	Maximum Units of Parallelism Assigned to a Single Processor	Predicted Speedup
1	15	1.000
2	8	1.875
3	5	3.000
4	4	3.750
5–7	3	5.000
8–14	2	7.500
15	1	15.000

This behavior is commonly seen with programs parallelized using OPENMP and its predecessors (it can also show up in other cases with a limited amount of parallelism [Bettge et al. 1999]). Providing that a program is able to meet its performance criteria, it is probably not appropriate to strongly penalize a program for this type of behavior. Instead, one should take this type of behavior into account when establishing the definition of ideal speedup.

This deviation from linear speedup is not an example of poor load balancing. Poor load balancing occurs when one or more processors receive significantly more work than the remaining processors. In this case, the distribution of work is limited by the limitations of integer arithmetic and, therefore, should be considered to be perfectly balanced (even though some processors might receive one more unit or work than another processor). Similarly, this is not an example of Amdahl’s law, since the loop is fully parallelized.

4. Hard Scalability

When discussing the actual scalability of a program, one really needs to talk about the combination of the program, the hardware, and the data set. The earliest metric for scalability is referred to as either hard scalability or fixed size scalability. This assumes that one has a fixed problem to solve and one wants to know how many processors are required to deliver an acceptable level of performance. There can be a number of reasons why the program will fail to deliver ideal speedup. Furthermore, on real distributed memory architectures running real codes and data sets, one frequently finds that large data sets cannot be run using a single processor of an **MPP** (most commonly due to insufficient memory). Smaller data sets that can be run on a single processor of an MPP may have a poor communication-to-computation ratio and, therefore, will show a low level of scalability. As a result of these problems, another metric was proposed, soft scalability, and it will be discussed in section 5.

Many of today's MPPs have powerful enough processors and enough memory per processor that enable many problems to be run on just one or two processors, if only for the purpose of running a scalability study. Therefore, let us briefly consider the three most commonly mentioned reasons for deviations from ideal speedup.

(1) Amdahl's Law: run time = serial run time + parallel run time. As the number of processors approach infinity, the parallel run time will asymptotically approach zero, and the run time will asymptotically approach the serial run time. Therefore, so long as one cannot eliminate the serial run time, there is an upper bound on speed at which a particular machine can run a particular job.

(2) Communication costs are nearly always a function of the number of processors being used. In some cases the function is a weak one (e.g., O(log(N))), while in other cases it can be much stronger (e.g., O(N)). This now gives us the following: parallel run time = parallel computation time + communication time. Therefore, even if the serial run time is zero, the run time will not asymptotically approach zero. Instead, a plot of the run time as a function of the number of processors used is expected to be U shaped. In other words, there is a small range of processors for which the level of performance will reach a maximum. Past that point, the performance will actually drop off as the number of processors is increased (Almasi and Gottlieb 1994). It is important to note that these costs are primarily a function of three things (the hardware, the number of messages [along with their distribution], and the size of the messages).

(3) The load balance: For example, if each part of an airplane's outer surface is assigned to a different processor, then one process would get most, if not all, of the fuselage. Each wing would be assigned to another processor, and, finally, the tail assembly would be assigned to a small number of processors. Assuming that all of the components are grided at the same resolution, then the processor with the fuselage might be performing upwards of 50% of the work. This would limit the potential for parallel speedup to no more than a factor of 2. Clearly a better approach is needed. Three commonly used approaches are:

(1) Domain decomposition, which breaks up the larger zones into more manageable pieces.

(2) Processing the zones one at a time and parallelizing the processing of the individual zones using loop-level parallelism or other techniques.

(3) Domain agglomeration, which would assign multiple zones to a single processor. This would be of little value in this case, but might be of value when all of the zones are small, but the range of zone sizes cannot be ignored. Recently, James Taft (a contractor for the NASA Ames Research facility) has been giving talks on some work that he has been doing in this area.

5. Soft Scalability

Soft scalability is also known as scaled speedup and was first proposed by J. L. Gustafson (1988). It proposes that so long as the run time of a job remains roughly constant when the job size and the number of processors increase at proportionally the same rate, then the job should be considered to be scalable. The advantage of this argument is that it allows one to get around the limitations imposed by Amdahl's Law. In fact, for many programs, it can eliminate both that limitation and problems with a poor ratio between communication and computation.

An excellent example of this approach at work was provided by Steve Schraml of the U.S. Army Research Laboratory (ARL), Aberdeen proving Ground, MD. When running CTH on the SGI R12K Origin 2000 and the SUN HPC 10000s located at the ARL-Major Shared Resource Center (MSRC), he measured the results in Table 2.

Table 2. The Scalability of the SGI R12000 Origin and the SUN HPC 10000 When Running CTH

System	Number of Processors	Grind Time in microseconds/zone/cycle		
		Measured	Predictions Based on Scaling	
			1 Processor Data	8 Processor Data
SGI Origin	1	36.979	36.979	N/A
	2	20.479	18.490	N/A
	4	10.355	9.2448	N/A
	8	7.2749	4.6224	7.2749
	16	4.0035	2.3112	3.6375
	32	2.0599	1.1556	1.8187
	48	1.4815	0.77040	1.2125
	64	1.2456	0.57780	0.90936
	96	0.73997	0.38520	0.60624
SUN HPC 10000	1	47.558	47.558	N/A
	2	25.622	23.779	N/A
	4	11.875	11.890	N/A
	8	7.0330	5.9448	7.0330
	16	3.7468	2.9724	3.5165
	32	1.8792	1.4862	1.7583
	48	1.2385	0.99079	1.1722
	60	1.1170	0.79263	0.93773
	63	1.1075	0.75489	0.89308
	64	1.1332	0.74309	0.87913

Two important objections to this approach are

(1) It doesn't address the problem of what to do if the speed at which problem A runs is unacceptable. Presumably, if one runs a problem N times larger using N times as many processors, the speed will still be unacceptable. The obvious answer is to use more processors for the current problem size. However, that raises the question of hard scalability. Potentially, this could result in some problems being run on so many processors that while their overall performance is good, their poor processor performance might be deemed to be unacceptable. This can be an especially bad problem if it causes one to run out of processors.

(2) This metric cannot be applied to any problem where the parallelism is not directly proportional to the problem size. In particular, when parallelizing the implicit **CFD** code F3D while using loop-level parallelism, it was discovered that for two important loops, there were dependencies in two out of three directions. Therefore, each of the dimensions of each zone is doubled, the amount of work increases by a factor of eight, while the parallelism increases by only a factor of two.

These can be important objections, since using the wrong metric or an inappropriate metric for the case at hand can lead to the wrong conclusions. In some cases, this might result in one choosing a suboptimal solution, while, in other cases, it might result in a project being abandoned entirely.

6. Throughput

While this metric is important to all users, it can be especially important to those users running parametric studies. These studies can be grouped into three categories:

(1) There are a large number of jobs to run, with no one job requiring a large number of resources. Furthermore, there are no dependencies between the runs, so one can, in theory, run all of them at the same time.

(2) There are a significant number of jobs to run, but they require a moderate-to-large amount of at least one resource (e.g., memory). However, there are few, if any, dependencies between the runs, so one may be able to run a limited number of these jobs at one time.

(3) There are a significant number of jobs to run, with no one job requiring a large number of resources. Unfortunately, there are dependencies between the runs, so one is again limited as to how many jobs can be run at one time.

The importance of these categories is that for a throughput optimized site, the first case may be able to achieve an acceptable level of performance while using a limited number of processors per job. In the other two cases, one will almost always want to use a larger number of processors per job. Therefore, in those cases, the scalability of the job takes on added importance.

7. Serial Efficiency

Most of this paper has dealt with the scalability. Now let us return to the question of serial efficiency. Even if one is running similar programs based on the same algorithm using similar parallelization strategies, differences in serial efficiency can significantly affect the performance of the programs. In particular, we will consider the performance of three versions of the F3D program that was previously mentioned. Marek Behr, formerly of the **AHPCRC**, produced two versions of the code designed to run on distributed memory platforms. One version used SHMEM calls and could be run on either the SGI Origin 2000 or the Cray T3E. The other version of this code used the more portable, but arguably less efficient **MPI** calls. The third version of the code was written by Daniel M. Pressel and was based on compiler directives for loop-level parallelism. As such, it could only be run on a shared memory platform and is highly dependent on the design characteristics of the platform being used. Table 3 contains results from running these codes on several different platforms for a 1-million grid point test case.

From the results in Table 2, one can see that there are a number of factors which can affect the performance of a program. The peak speed of the processor and the number of processors used are only two of those factors.

Table 3. The Performance of Various Versions of the F3D Code When Run on Modern Scalable Systems

System	Peak Processor Speed (MFLOPS)	Number of Processors Used	Version	Speed (time steps/hour)
SGI R12K Origin 2000	390	8	Compiler Dir.	793
	600		SHMEM	382
SGI R12K Origin 2000	390	32	Compiler Dir.	2138
	600		SHMEM	989
	600		Compiler Dir.	2877
SGI R12K Origin 2000	390	48	Compiler Dir.	2725
	600		SHMEM	1083
	600		Compiler Dir.	3545
SGI R12K Origin 2000	390	64	Compiler Dir.	2601
	600		SHMEM	1050
	600		Compiler Dir.	3694
SGI R12K Origin 2000	390	88	Compiler Dir.	3619
	600		SHMEM	1320
	600		Compiler Dir.	5087
Cray T3E-1200	1200	8	SHMEM	349
Cray T3E-1200	1200	32	SHMEM	1062
Cray T3E-1200	1200	48	SHMEM	1431
Cray T3E-1200	1200	64	SHMEM	1705
Cray T3E-1200	1200	88	SHMEM	2443
Cray T3E-1200	1200	128	SHMEM	2948
IBM SP (160 MHz)	640	8	MPI	199
IBM SP (160 MHz)	640	32	MPI	342
IBM SP (160 MHz)	640	48	MPI	420
IBM SP (160 MHz)	640	64	MPI	423
IBM SP (160 MHz)	640	88	MPI	396
SUN HPC 10000	800	8	Compiler Dir.	999
SUN HPC 10000	800	32	Compiler Dir.	2619
SUN HPC 10000	800	48	Compiler Dir.	3093
SUN HPC 10000	800	56	Compiler Dir.	3391
SUN HPC 10000	800	64	Compiler Dir.	2819
HP V-Class	1760	8	Compiler Dir.	1632
HP V-Class	1760	14	Compiler Dir.	2392

8. Conclusions

This paper has discussed a number of issues relating to the topics of scalability and performance. It has been shown that for some problems, the ideal speedup will resemble a stair step rather than a straight line. With this concept in hand, two ways for measuring scalability were discussed, with emphasis placed on their strengths and weaknesses. This discussion included examples using these metrics. Hopefully, this paper will help the reader in his/her work. In particular, it points out that while scalability is good, most users are concerned with performance and throughput.

Acknowledgements

The author would like to thank Steve Schraml and Marek Behr for permission to use their results in this paper. He would also like to thank his many colleagues that have graciously assisted him in all aspects of the preparation of this paper.

Glossary

AHPCRC -	Army High Performance Computing Research Center
CFD -	Computational fluid dynamics
MFLOPS -	Million floating point operations per second
MIMD -	Multiple instruction multiple data
MPI -	Message passing interface
MPP -	Massively parallel processor
RISC -	Reduced instruction set computer
SIMD -	Single instruction multiple data

References

Almasi, G. S., and A. Gottlieb. *Highly Parallel Computing 2nd Edition*, Redwood City, CA: Benjamin/Cummings Publishing Company, Inc., 1994.

Bailey, D. H. “RISC Microprocessors and Scientific Computing.” *Proceedings for Supercomputing 93*, 1993.

Bailey, F. R., and H. D. Simon. “Future Directions in Computing and CFD.” *American Institute of Aeronautics and Astronautics, Inc.*, http://www.nas.nasa.gov/NAS/TechReports/RNRreports/hsimon/RNR-92-019/RNR-92-019.o.html, 1992.

Bettge, T., A. Craig, R. James, W. G. Strand, Jr., and V. Wayland. "Performance of the PCM on the SGI Origin 2000 and the Cray T3E." *The 41st Cray User Group Conference*, Minneapolis, Minnesota, May 1999.

Gustafson, J. L. "Reevaluating Amdahl's Law." *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, The Association for Computing Machinery, Inc., May 1988.

Mascagni, M. "Parallel Wiener Integral Methods for Elliptic Boundary Value Problems: A Tale of Two Architectures." http://sushi.st.usm.edu/~mascagni/ftp/astfalk.ps, originally published in 1990 in SIAM News, vol. 23, no. 4, July 1990.

Oberlins, S. Keynote slides for ISCA'99. *The 26th International Symposium on Computer Architecture*, http://www.neci.nj.nec.com/isca99/.

Simon, H. D., and L. Dagum. "Experience in Using SIMD and MIMD Parallelism for Computational Fluid Dynamics." http://www.nas.nasa.gov/NAS/TechReports/RNRreports/hsimon/RNR-91-014/RNR-91-O14.o.html, 1991.

Simon, H. D., W. R. Van Dalsem, and L. Dagum. "Parallel Computational Fluid Dynamics: Current Status and Future Requirements." http://www.nas.nasa.gov/NAS/TechReports/RNRreports/hsimon/RNR-92-004/RNR-92-004.html, 1994.

Singh, K. P., B. Uthup, and L. Ravishanker. "Parallelization of Euler and N-S Code on 32 Node Parallel Super Computer PACE+." Presented at the ADA/DRDO-DERA Workshop on CFD, 1998.

Wang, G., and D. K. Tafti. "Performance Enhancement on Microprocessors With Hierarchical Memory Systems for Solving Large Sparse Linear Systems." *The International Journal of Supercomputing Applications*, February 1997.